LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-207651

A Task-parallel Clustering Algorithm for Structured AMR

B. T. N. Gunney, A. M. Wissink

November 2, 2004

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

A Task-Parallel Clustering Algorithm for Structured AMR

Brian T. N. Gunney
Lawrence Livermore National Lab
gunneyb@llnl.gov

Andrew M. Wissink
Lawrence Livermore National Lab
awissink@llnl.gov

November 2, 2004

Abstract

A new parallel algorithm, based on the Berger-Rigoutsos algorithm for clustering grid points into logically rectangular regions, is presented. The clustering operation is frequently performed in the dynamic gridding steps of structured adaptive mesh refinement (SAMR) calculations. A previous study revealed that although the cost of clustering is generally insignificant for smaller problems run on relatively few processors, the algorithm scaled inefficiently in parallel and its cost grows with problem size. Hence, it can become significant for large scale problems run on very large parallel machines, such as the new BlueGene system (which has $O(10^4)$ processors). We propose a new task-parallel algorithm designed to reduce communication wait times. Performance was assessed using dynamic SAMR re-gridding operations on up to 16K processors of currently available computers at Lawrence Livermore National Laboratory. The new algorithm was shown to be up to an order of magnitude faster than the baseline algorithm and had better scaling trends.

1 Introduction

Adaptive mesh refinement (AMR) is an approach for discretizing and solving science and engineering problems on computational meshes. It is useful for problems with localized fine-scale regions in the computational domain. By placing the mesh points and computational efforts where they are needed most, AMR can require significantly less computational resources than using uniformly fine meshes. In a dynamic problem where solution features move and appear or disappear, the AMR mesh changes to adapt to the changing features. Grid points can be automatically inserted and removed where needed.

Structured AMR (SAMR) is an approach originally proposed by Berger, Oliger, and Colella [BO84, BC89] that composes the adaptively refined mesh by overlaying successively finer individual structured grids where higher resolution is needed (figure 1). The mesh is composed of a sequence of levels, each having

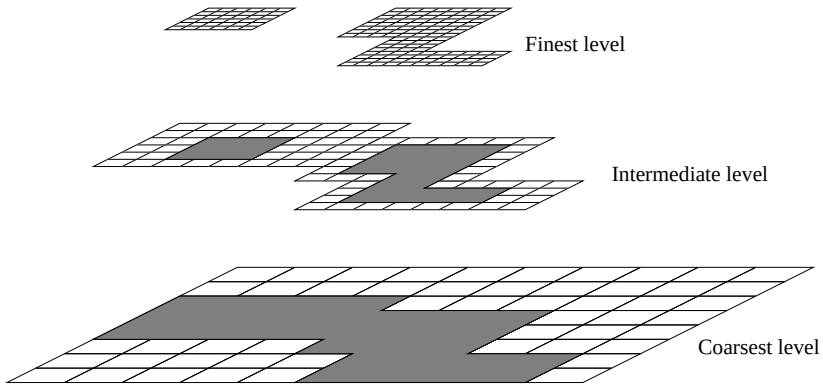


Figure 1: Simple 2D structured AMR mesh with three levels of refinement. Shaded cells are those where finer grids are overlaid.

greater resolution than the previous. As shown in figure 1, fine grids are aligned with coarse grid lines, meaning that coarse grid lines coincide with fine grid lines at fixed intervals where the fine grid overlays the coarse grid.

The clustering algorithms described in this paper are used during the dynamic gridding steps in the SAMR implementation. SAMR mesh adaptivity consists of first replacing the current level with an updated one and, second, transferring data to the new level. Refinement consists of adding a finer level to the hierarchy, overlaying the finest existing level. Each of these operations builds new levels. Clustering generates the initial set of *boxes* from which to build the new levels. The boxes are logically rectangular, defined by the indices of its lower and upper corners.

To create a new level, the application must determine what regions the new level should cover and generate the structured grids to cover them. A feature detection scheme specific to the problem is commonly used to "tag" cells that the new level should cover, e.g., it finds cells that contain large gradients or numerical error. The clustering algorithm finds a set of boxes covering the tagged cells and preferably few untagged cells. Each box is considered a cluster of tagged cells. The set of boxes may be further processed (e.g., to enforce size constraints). The grids in the new level is created directly from the final set of boxes.

A common algorithm used in SAMR for the clustering step is the algorithm proposed by Berger and Rigoutsos [BR91]. This algorithm is generally quite fast and works well in serial and in parallel with moderate numbers of processors. However, its scaling properties can be poor, and it can be expensive for large problems run on many processors [WHH03]. This paper describes a new algorithm, based on that of [BR91] and intended to scale better on large parallel computers.

It is important to note that clustering is only done when a new grid level is

generated so that the frequency of grid adaptation affects the overall clustering cost. For example, a problem with static adaptivity will only perform clustering once to generate the initial refined grid. While it is possible to reduce the overall clustering costs in a computation by reducing the frequency with which grids are adapted, this approach introduces other overheads. For instance, the refinement region must include extra buffer zones to assure dynamic features in the solution do not move beyond the refined region between refinement steps. In this work we focus on improving the efficiency and scalability of the clustering operation itself, independent of how often it is applied.

The algorithms described are implemented in the SAMRAI framework [HK02, SAM04] developed at Lawrence Livermore National Laboratory, though the concepts we develop should apply to other SAMR implementations.

We review the Berger-Rigoutsos clustering algorithm in the next section. Section 3 discusses the performance of current parallel versions of the algorithm and motivates the need for improved efficiency. The new task-parallel algorithm is described in section 4. Section 5 contains the performance results.

2 Previous Clustering Algorithms for SAMR

In 1991, Berger and Rigoutsos [BR91] considered a number of general variations of *bottom-up* and *top-down* clustering algorithms for SAMR. Bottom-up variations start with seed points computed from the tagged-cell pattern and build boxes around the seed points using variances of k-means partitioning algorithm [And73, Har73]. The top-down algorithm places all tagged cells into an initial single box then splits the initial and subsequent boxes to eventually form the final set of boxes (see figure 2). These variations are forms of hierarchical clustering. Each hierarchical clustering corresponds to a tree, known as a *dendogram*, with the initial grouping at the root and the final groupings at the leaves [DH73] (see figure 3).

The best algorithm found in [BR91] was a top-down algorithm. The criteria for whether and where to split a cluster are based on ideas from edge detection algorithms [MH80], using *signatures*. Signatures for a d-dimensional box are computed by projecting each tag to the d axes and summing the number of tags at each point on the axes (see figure 2a). The signatures form one-dimensional descriptions of the tag distribution in higher dimensional boxes.

Signatures are used to decide whether and where to split a candidate box in the top-down algorithm, according to the following criteria:

1. A box is split if it does not meet a preset *efficiency threshold*. Efficiency is defined as the ratio of the number of tagged cells in the box to all cells in the box. It controls the degree of extra refinement in untagged regions.

2. The first preferred location to split a box is at a hole, or zero value, in a signature.

3. If no hole is found in the signature, the next preferred cut location is at an inflection point (zero-crossing of the second derivative) of a signature.

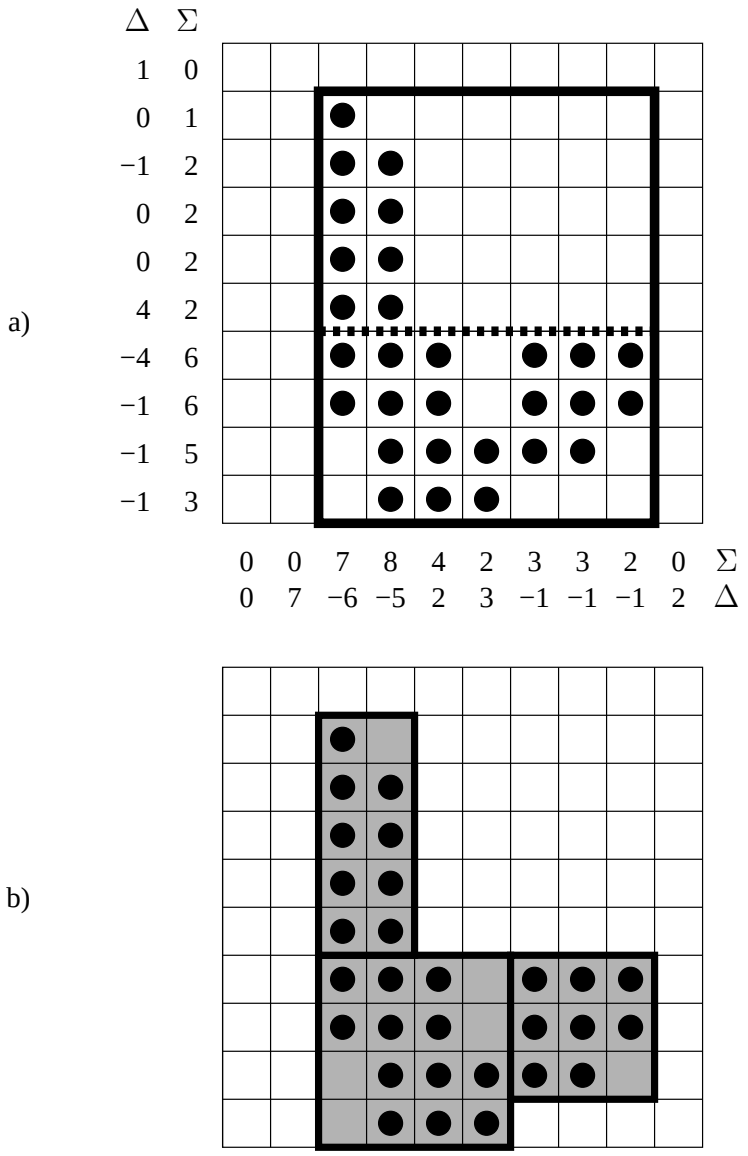


Figure 2: A top-down hierarchical clustering example using the Berger-Rigoutsos algorithm [BR91]. Tagged cells are marked by dots. a) Signatures, bounding box and cut used by the Berger-Rigoutsos algorithm. Σ is the signature. Δ is the undivided Laplacian of the signature. Heavy-lined box is the bounding box of the clustered tags. Dashed line is the location of the cut based on the inflection point criterion (see text). b) Resulting box clusters.

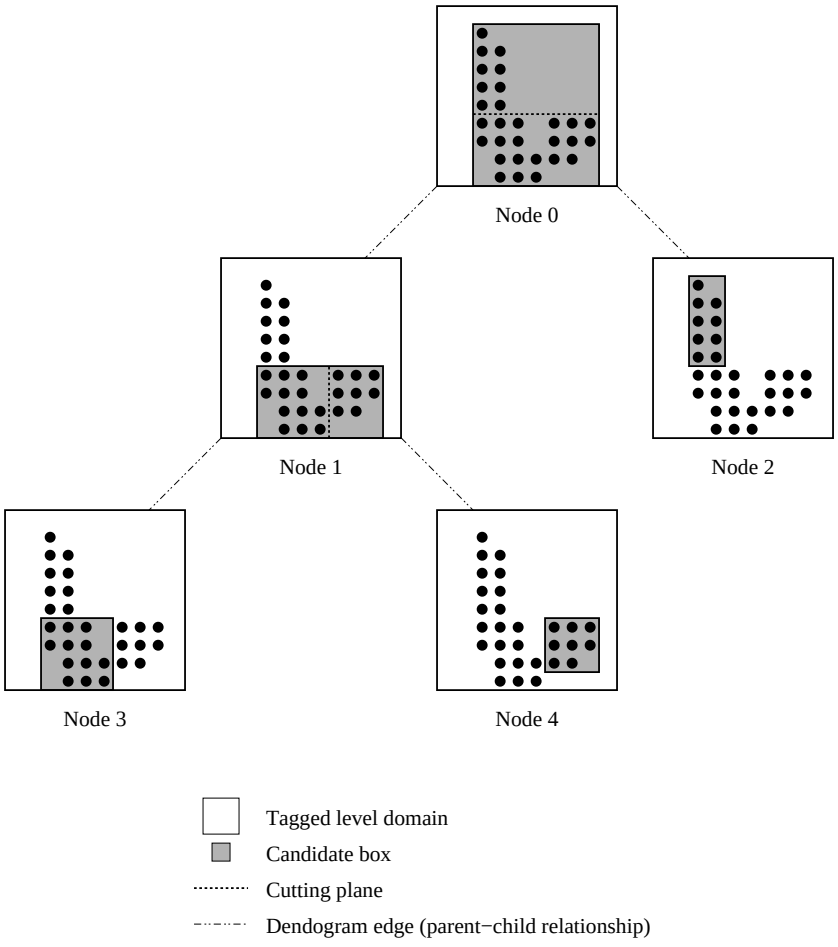


Figure 3: Dendogram corresponding to the clustering example in figure 2. Node 0 is the initial cluster. Nodes 2, 3, and 4 comprise the final clusters.

Figure 2a shows the second derivative of the signature approximated by the undivided Laplacian Δ.

The top-down algorithm is outlined as follows in [BR91]:

Algorithm 2.1: BergerRigoutsos(*boxes*)

$i \leftarrow 1$
while ($i <$number of boxes)
do
- **if** efficiency of box $B_i <$threshold
  - **then**
    - compute signatures
    - find the best place to split box B_i
    - **if** found a place to split
      - **then**
        - split box B_i in two
        - append new boxes to end of list *boxes*
      - **else** $i \leftarrow i + 1$ (go to next box)
  - **else** $i \leftarrow i + 1$ (consider next box)

If the efficiency threshold is set to 1, every new box constructed will contain only tagged cells. While this may seem desirable, in practice it leads to construction of many small boxes, a process that that introduces other overheads. It is generally most efficient to set the threshold to something slightly less than 1 which reduces the number of boxes but includes some cells in the refined region that were not originally tagged to be refined.

Rantakokko [Ran03] described a similarly structured top-down algorithm but (optionally) with specific criteria for choosing the next box to split. Whereas [BR91] equivalently chooses the next box from a breadth-first search of the current dendogram leaves, [Ran03] chooses, from all current dendogram leaves, the one that has the most untagged cells. Different criteria are also used to determine the cutting plane, with the goal of minimizing the number of tagged cells in the cutting plane.

SAMRAI uses a slight variation of algorithm 2.1, employing recursive function calls in place of a loop. Each recursion takes a single candidate box and builds up a list of cluster boxes for the tags in the candidate box. If a candidate box is split, the recursive function is called for the children boxes. This is equivalent to using a depth-first search for the next candidate box to analyze, but it yields the same set of boxes as algorithm 2.1 would. The SAMRAI clustering algorithm is:

Algorithm 2.2: RecursiveBergerRigoutsos(C, b)

comment: Compute a set of boxes C (the cluster), starting with candidate box b

compute signatures of b
$b \leftarrow$bounding box of tags in b
if efficiency$(b) \geq$ threshold
then $C \leftarrow \{b\}$
else
- split b in two to create boxes b_L and b_R
- RecursiveBergerRigoutsos(C_L, b_L)
- RecursiveBergerRigoutsos(C_R, b_R)
- **if** (length$(C_L) > 1$ or length$(C_R) > 1$ or combined efficiency $> \beta$ efficiency(b)) (i)
  - **then** $C \leftarrow \{C_L, C_R\}$
  - **else** $C \leftarrow \{b\}$

Algorithm 2.2 adds at statement (i) an additional step of checking the *combined efficiency* of the boxes returned by the recursions. The combined efficiency is defined as the ratio of tagged cells summed in the left and right boxes to the sum total of cells in those boxes (this ratio is defined only if the left and right boxes are not split). If the combined efficiency does not improve by the factor β over the parent's efficiency, the parent's box is taken instead. The goal of recombining is to avoid fragmented boxes if the potential gain in efficiency is minimal. The recursive structure of algorithm 2.2 provides the parent-children and sibling relationships needed for the check on combined efficiency. Such a check may be more difficult to implement in algorithm 2.1, where the parent is discarded when the children are created.

Although the structure of algorithms 2.1 and 2.2 differ slightly, the graph of the flow of the implementation (i.e., the dendogram–see figure 3) is identical for both. They use the same method to evaluate the candidate box and either accept it or determine where to split it. This general method is applied to each *node* (i.e., box) in the dendogram of the clustering. We refer to this general method as the *node routine* herein. Since boxes and nodes have a one-to-one correspondence, we use the terms interchangeably.

3 Parallel Performance of the Clustering Algorithm

Clustering involves accumulating and operating on tagged cells from the entire problem domain. Hence, its cost grows with problem size. For relatively small problems, run on $O(10^2)$ or fewer processors, clustering is usually quite fast and its cost is generally negligible. However, as we move to larger scale problems

run on many processors the cost becomes more significant. In [WHH03], a set of benchmark adaptive problems run on up to 1024 processors shows that the cost of clustering grows significantly with number of processors, even though the problem size remains fixed. The increasing cost comes from the additional communication required for the collective operations on the tagged cells. Because the cost of clustering increases with increasing problem size *and* increasing number of processors, the combined effect of running large problems on large numbers of processors can become a significant bottleneck.

Like most SAMR implementations [CGL+03, BB87, KB96, RBL+00], SAMRAI uses a domain decomposition single-program multiple-data (SPMD) parallelization approach. Individual structured grids that form a level are distributed to processors in such a way that the load is balanced. All data on a grid resides on the processor that owns the grid. When clustering with this distributed data model, the signature computation requires communication. This can be done using standard global sum operations; partial signatures are computed using the data local to each processor and the global signature is formed by summing the contributions from all processors. Because computing a signature requires numerous sums (one for each point on each axis) and many signatures are computed during the clustering operation (a new signature must be computed for each candidate box) many global sums will be performed, which can become expensive.

A simple way in which the clustering can be implemented on a parallel system is for each processor to compute a partial signature using just the tags data that it owns. The standard collective function MPI_Allreduce is used to sum the partial signatures to compute the full signatures and return it to all processors. With this call, every processor gets the sum data and can proceed with the otherwise unchanged algorithm. An inefficiency in the use of global sums is that it often involves much needless communication with processors that do not hold data used in the full signature. The clustering algorithm initially starts with a box that covers the entire domain, so every processor can potentially have tag data to contribute to the initial signature. However, once this initial box is split, tagged cell data on each of the sub-boxes can only come from processors that hold data in the child-box region. As the recursion continues, the candidate boxes get smaller and smaller, and although fewer and fewer processors are needed to compute the complete signature, each global sum still requires messages from all processors. An alternative implementation that uses a local signature reduction is presented in [WHH03]. The key feature in this implementation is that it utilizes subsets of processors to perform the communication only among the processors that hold data required for the signature computation. One processor is designated as a *manager* (e.g., processor zero) and any processor that holds data on the particular box being processed is designated as a *worker*. A processor is considered a worker if the grids it owns on the *tagged level* (the level where the cells are tagged) grids that overlap the candidate box. The worker processors compute their local signatures and send this information to the manager. The manager accumulates the results from the workers, computes the global signature, decides whether and where to split

the box, and sends the decision to the workers. A new set of worker processors is designated for each of the sub-boxes, and the process is repeated. At the end of the recursion the set of boxes computed by the manager is broadcast to all processors.

From an algorithmic standpoint the global signature reduction and local signature reduction implementations generate identical output sets of boxes. Because less communication is required, the local reduction implementation was found to be significantly faster on greater than $O(10^2)$ processors.

In spite of the performance gains from using local signature reduction, the algorithm is still limited in its scalability for two reasons. First, the slowest part of this algorithm is the communication, which tends to worsen instead of improve as the number of processors increases. Second, it still relies on a single manager processor that synchronously coordinates contributions from multiple worker processors. In order to effectively scale on systems with $O(10^5)$ processors we need to pursue alternative implementations.

4 Task-parallel Clustering Algorithm

Algorithms 2.1 and 2.2 described in section 3 follow a SPMD (or data-parallel) model. Operations on distributed data (computing local signatures, for instance) were parallelized, but operations on collective data (the full signatures, group formations, etc.) remained sequential. Therefore, the overall algorithm retained sequential logic. The top-down hierarchical approach opened an opportunity to integrate task-parallelism into the logic. After a box was split, the left and right branches formed tasks that were mutually independent, because the decisions for one box is completely independent on the decisions for the other. In figure 3 for instance, nodes 1 and 2 were mutually independent, as were nodes 3 and 4. If node 1 was split before node 2 was completed, then nodes 2, 3 and 4 were all mutually independent. It was possible to work on the independent nodes concurrently rather than sequentially[1] as was done in algorithms 2.1 and 2.2.

In the task-parallel approach, each instance of the node routine made a natural task. Each task was data-parallel, requiring communication within the group of processors holding data for the box. We could significantly reduce the overall time spent waiting on communication by switching to another task when the current task is waiting. We use task and node interchangeably, because there is a one-to-one correspondence between them.

To have concurrent tasks, we must know which are independent. Sibling tasks were independent because the decisions on one box did not affect the sibling box. However, parents and their children were not independent. A child task was dependent on its parent because the child could not be created until

[1] Although it was tempting to partition the computer to work on the left and right branches in parallel, this did not appear to be feasible because (1) branching and the location of the cut was not known *a priori* and (2) some processors might still have to participate in both branches.

the parent is split. A parent task depended on the results of its children so it can perform the combined efficiency check. At any point in the above top-down clustering algorithms, nodes that have no children, and whose children have all completed running, are independent and can proceed concurrently. Nodes with active children are dependent on the results of all its children and must wait until they are finished.

Concurrent tasks could be performed in a manner similar to multi-threading, where each task corresponded to one thread. Task switching was implemented by exiting one task at some point in the node routine and starting (or restarting) a different task.

In the rest of this section, we describe a mechanism for exiting a task before it completes, a mechanism for selecting which task to run and further details about the implementation.

To exit a task before it completes, we implemented a *self-suspending node routine* by building in logic for suspending the routine and returning to where it left off. The node routine suspended itself by storing its state in a data structure and exiting. When the node routine was restarted, it jumped to the point where it was suspended, using the stored information. We chose places where the processor had to wait as points where the node routine would suspend itself. The node routine could be viewed as a sequence of alternating local computation and wait phases, much like other SPMD applications. A node might wait for its communications or for its children to complete. (Communications in each node routine must be initiated by non-blocking calls so that the task is not forced to wait for the communication to finish.) Algorithm 4.1 illustrates the general logic for implementing the self-suspending node routine.

Algorithm 4.1: SELFSUSPENDINGNODEROUTINE(*node*)

comment: *node* is a data structure containing data associated with a particular dendogram node. This data includes the state of the node routine when it was suspended.

if starting routine
  then go to 0 (i)
if reducing signatures
  then go to 1 (ii)
if ...
  then go to ...

0: (iii)
compute local signatures for *node*
initiate sum-reduce operation for signatures
1: (iv)
if sum-reduce is incomplete
  then { PUTINTASKMANAGER(*node*) (v)
           return
box ←bounding box of tags
...
return

The "go to" statement (i) directs the routine to the label at statement (iii), similarly for statements (ii) and (iv). At statement (v), where the node routine would return before it completes, it places itself back in the task manager for restarting at a later time.

The node routines were driven to completion by a *task manager*, which was essentially a user-space thread controller, specialized for tasks that had distinct computation and wait phases. The role of the task manager was to keep track of waiting tasks and eventually restarting them. When a node routine suspended itself, it returned control to the task manager which picked the next task to start (or restart). Many concurrent tasks could exist in the task manager, and each time a node is split, two more tasks are generated. The manager was responsible for allowing all tasks to run in turn. The algorithm is completed when no more tasks are in the manager

A simple manager could be built around a queue, Q, of unfinished tasks, as shown in algorithm 4.2. This manager takes the first task from the queue and restarts it. If the node routine suspended itself before completing, as described in algorithm 4.1, it would have placed itself back in the queue to be restarted at a later time. The task manager would call SELFSUSPENDINGNODEROUTINE on each node as many times as it takes to complete it.

Algorithm 4.2: QUEUETASKMANAGER(Q)

comment: Q is a queue (list) of tasks.

while $length(Q) > 0$

do
- $node \leftarrow$ DEQUEUE(Q)
- SELFSUSPENDINGNODEROUTINE($node$)
- **comment:** If $node$ did not complete, it would have been put back in the queue by the self-suspending node routine of algorithm 4.1.

The queue would start with the root node of the dendogram. Though not explicitly shown in algorithm 4.1, children nodes were placed in the manager as they are generated rather than recursively entered by their parents as is done in algorithm 2.2. The queue task manager is loosely similar to the outer loop of algorithm 2.1. A more sophisticated task manager is described in section 4.2.

In addition to the groups of processors that participate in evaluating a given node, we chose from the node's participating group an *owner* processor. The owner became the root of communications in the group, getting collective data from the group, making decisions based on the collective data and sending the decisions back to the group. In [WHH03], a single processor (e.g., processor id 0) participates in all groups and is the designated owner of every group. We called this the *single-owner* mode. By choosing different owners, we could relieve the bottleneck (both computation and communication) that occurs at the single owner processor. Choosing different different owners leads to the *multi-owner* mode. (Our "owner" corresponds to the "manager" in [WHH03]. We use the term "owner" to avoid confusion with the task manager.)

Other algorithms in SAMRAI required that the full output (the accumulated results from all node routines) be given to all processors, requiring a collective communication after all node routines complete. At the end of each node routine, only the owner had the final box. In algorithm 2.2 and in the single-owner mode of the new algorithm, one processor has all the output and only needs to broadcast it at the end of the clustering step. One issue that arises in the multi-owner mode is that clustering results were distributed over the multiple owners. Getting this output to all processors required an all-gather communication, which is slower than the simple broadcast used in the single-owner mode.

Some additional data must be stored for the implementation of the task-parallel algorithm. In the node data structure of algorithm 4.1, the state of the node (its box, its group, its owner, where the routine left off, etc.) was saved so that the node could be restarted as if it had not been interrupted. We added references to parent and children and data supporting communications to this data structure. The references to a node's parent and children were used to put parents and children into the task manager, check on their states, etc. Data supporting communication included those for of a tree-based collective commu-

nication scheme, similar to that described in [WHH03]. While it is possible to use MPI communication groups for collective communication, we determined in [WHH03] that collective communications using the tree were more efficient than creating an MPI communicator for each of the many groups. In the tree-based scheme, each processor in a group is assigned a node on the tree, with the owner processor at the root. Broadcasts from the root sent the messages toward the leaves. Reductions and gathers sent the messages toward the root. We explicitly implemented a *balanced tree*, in contrast to [WHH03], where balanced trees are not assured.

To support the multi-owner mode, an additional communication step was required in the node routine. For each node we defined the *dropout group* as the set of processors that participated in the node's parent but not in the node itself (because they do not hold any data for the node's box). In the multi-owner mode, the owner for each node broadcasts the node's final result to the node's dropout group. Although processors in the dropout group did not participate in evaluating a child node, they needed the node's final results in order to perform the combined efficiency check for the node's parent.

4.1 Task-parallel Node Routine

In this section we fill in more detail about the node routine for the task-parallel algorithm. For readability, we omit the details of suspending and restarting, which were already shown in algorithm 4.1.

The new node routine is shown in algorithm 4.3. Functions attributable to a node use the node's internal data and were written using the node's dot (.) operator borrowed from C++ syntax. For example, *node*.Box() returned the box of the node, *node*.Overlap() returned amount of overlap between the candidate box and local grids on the tagged level, and *node*.ComputeLocalSignatures() computed and stored the node's local signatures from the local tag data.

Algorithm 4.3: TaskParallelNodeRoutine(*node*)

```
if node.Overlap() > 0                                                    (i)
  then { node.ComputeLocalSignatures()
         node.SumReduceSignatures()
         if node.Owner() = local process rank
           then { node.Box() ← bounding box of tags
                  node.AcceptOrSplit()
         node.BroadcastAcceptability()
         if node.Acceptability() = false                                 (ii)
           then { node.CreateChildren()
                  node.FormChildGroupsAndOwners()
                  PutInTaskManager(node.LeftChild())
                  PutInTaskManager(node.RightChild())
                  node.WaitForChildren()                                 (iii)
                  node.CheckCombinedEfficiency()

comment: Owner broadcasts results to dropout group.
         (See text.)
if { node.Owner() = local process id  or
   { node.Overlap() = 0                                                  (iv)
  then {node.BroadcastToDropouts()

comment: After last child completes,
         parent may continue.
if node.Parent() ≠ 0 and node.Sibling() is completed
  then PutInTaskManager(node.Parent())                                   (v)

return ()
```

In algorithm 4.3, the if-block at line (i) was executed by the participating group and largely followed the steps in algorithms 2.1 and 2.2. The processors computed their local signatures and and sum the signatures on the owner processor. The owner decided to accept or split the candidate box and broadcast the decision to the participating group (along with candidate boxes for children, if needed). If the box was not accepted, line (ii), the routine created children nodes and computed their groups. To select the owner, we chose the processor with the greatest overlap with the candidate box. The children were placed in the task manager so their node routines can eventually be run.

After putting the children in the task manager, the node must wait for them to finish. The function WaitForChildren called at (iii) was a wait similar but not identical to those caused by communications. The operational difference was that the node was not immediately put in the task manager. It was put there only after its last child finishes, at line (v). Differentiating communication waits

from children waits prevented a parent node from being unnecessarily checked until all its children are completed. When the parent continued, the method CheckCombinedEfficiency performed the same combined efficiency check done by algorithm 2.2 to see if the efficiency of the children warrants keeping them over their parent.

The middle block, line (iv), in algorithm 4.3 was entered by the node's owner and the dropout group. As discussed above, a child's dropout group contains those processors that participated in the parent's computation but do not participate in the child's. In BroadcastToNonOverlapGroup, the owner broadcasts the results of the participating group to the dropout-group. This step assures that the node results were available on all processors so that the parent of the node could perform the combined efficiency check.

4.2 Task Manager Algorithm

Some latitude existed in how the task manager selected the next task to check and, possibly, to restart. The simple queue task manager (algorithm 4.2) was inefficient as it went through an enormous number of cycles before finding a node with completed communications, consuming CPU cycles while making no progress. Instead of cycling through all waiting nodes, we would like to select those nodes that can make immediate progress. Only when these are exhausted would we wait on communications.

To quickly find nodes that can make immediate progress, we treated them differently from other tasks. We made queue-based task manager the first stage in a two-stage approach. The queue accepted only tasks that could make immediate progress. These were new tasks and tasks whose children had just completed. They were explicitly placed on the queue by their parent or child. Tasks that were waiting for communication were processed by a second stage that is entered when the queue is empty. Tasks waiting for communication were referenced through their `MPI_Request` objects, by mapping the requests back to the tasks. (`MPI_Request` objects are handles, returned by MPI, that refer to specific outstanding communication requests.) When a communication request completes, the task waiting for it were identified using the map and restarted. Communication waits were handled by MPI_Waitsome and invoked only when there were no more tasks that could make immediate progress, which effectively was when the queue was empty. We note that parent tasks waiting for their children to complete were left out of the two-stage task manager. We relied on children tasks to put their parent back into the manager when they completed (see algorithm 4.3).

Implementation of the two-stage task manager required storing a queue of tasks, an array of `MPI_Request` objects, and a mapping from the request objects to the tasks waiting for them. The array of request objects were input to MPI_Waitsome to get the next set of MPI communications that have completed. The algorithm follows:

Algorithm 4.4: TwoStageTaskManager()

while { There are tasks in the queue **or** there are incomplete communications
do {
  Restart all tasks in the queue until the queue is empty
  Wait for some communication calls to finish
  Restart tasks corresponding to completed communications

The outermost loop in algorithm 4.4 was required because as it executes, more tasks might be added to the queue and more communication operations might be launched.

5 Results

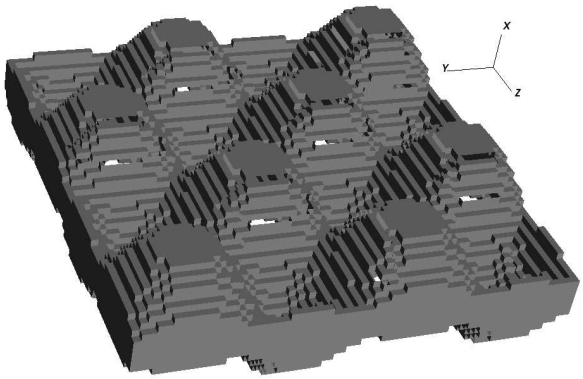
We evaluated the performance of the task-parallel top-down algorithm for a representative moving-grid simulation. Calculations are performed on up to 16K processors on four different parallel computer systems. Details of the problem, a description of the parallel systems, and timing results are presented in the remaining sub-sections.

5.1 Problem description

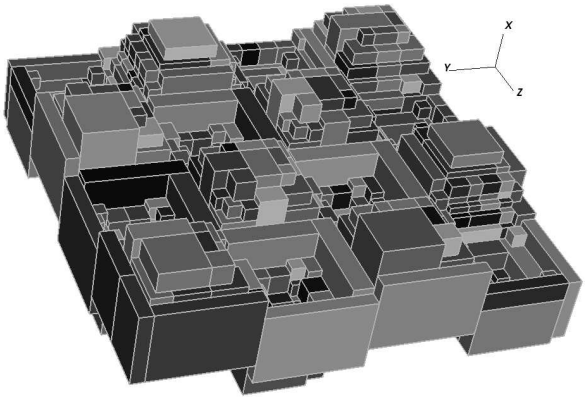
Performance is measured for clustering a 3D sinusoidal tagged cell distribution that advects through the domain (see figure 5.1). This problem does not perform a physics calculation but is representative of a geometrically complex shock wave moving through a domain. We leave out the physics to isolate the performance of the clustering operation alone. The grid domain size is 24x16x16 cells on the coarsest level. In physical space, the domain is a right hexahedron with a corner at (0,0,0) and the opposite corner at (3,2,2). The front is initially centered at (0.5, 0, 0) and moved (0.02, 0.005, 0.005) each time step. The hierarchy has 4 levels, with a refinement ratio of 2 for each level. The sinusoidal front moves through the domain at the above fixed velocity and the grids around it are regenerated from tagged cells five times. The problem size remains fixed with the number of processors. Multiple timings are done on each processor partition and the results presented are average times taken from several runs.

5.2 Parallel Systems

We present the results on the three current LLNL production parallel platforms and the new BlueGene/Light (BG/L) system. All are distributed memory parallel, with modest shared memory parallelism. Table 5.2 shows the characteristics of the platforms. The "Max number of processors" shown in table 5.2 were the numbers available for the experiments, not the absolute maximum on the platforms. Although the BG/L system would eventually have 64K processors, the full machine was not yet fully constructed, so only 16K processors were available.



Tagged cells.



Patches generated around the tagged cells.

Figure 4: Sinusoidal front test problem.

Machine name	Model	Processor type	Max number of processors	CPUs per computing node	Network type
Frost	IBM SP2	375 MHz Power 3	256	16	SP switch
MCR	Linux cluster	2.4 GHz Xeon	2048	2	Quadrics QsNet
Thunder	Linux cluster	1.4 GHz Itanium 2	2048	4	Quadrics QsNet
BG/L	Linux cluster	700 MHz PPC 400	16K	2	3D Torus

Table 1: Computers used to evaluate task-parallel algorithm.

5.3 Timing Results

During testing of the task-parallel algorithm we experimented with a number of different options and settled on four primary "modes" for which the algorithm showed interesting or optimal performance under different circumstances on different systems. These four modes were designated as:

- **Baseline** - the baseline mode used the implementation of algorithm 2.2 in section 2.
- **Balanced-tree Synchronous** - algorithmically identical to the "baseline" mode but used balanced trees for group communications. Two communication phases were completed in each task, 1) to reduce the signatures, and 2) to broadcast the acceptance. A single manager processor coordinated the algorithm and broadcasted the result at completion.
- **Single-owner Task-parallel** - task parallel implementation with a single owner. Like the synchronous mode, this used the same processor to accumulate the results from each group of processors operating on a node. Unlike the synchronous case, however, this mode used the node routine suspension to minimize time spent waiting for the communications. Like the "balanced-tree synchronous" mode, this mode required two communication phases per task and a single broadcast at the end to globalize the output.
- **Multi-owner Task-parallel** - task parallel implementation with multiple owners. This mode required one additional communication at each task (totaling three) to broadcast results to the dropout group of processors. After clustering, an all-gather communication was done to globalize the output. Although the node routines incurred one extra communication step per task and the slower all-gather at the end, the better load balance of the extra work done by owners sometimes made up for the additional cost.

Figure 5 shows the clustering times for the baseline and new task-parallel algorithms applied to the moving sinusoidal front problem run on various processor configurations of the four platforms tested. All timings shown were the maximum across all processors. The "baseline" algorithm 2.2 tested in [WHH03], clearly scaled poorly beyond 128 processors on all three platforms. On the IBM SP system, the "multi-owner task-parallel" mode was clearly fastest, whereas on the two Linux cluster systems, the "single-owner task-parallel" mode was fastest. It should be noted that on the Linux systems the synchronous algorithm using the balanced tree communication scheme was considerably faster than the similar baseline algorithm which did not use balanced trees, implying that use of smarter communication strategies could significantly enhance the performance of the synchronous approach. The task-parallel algorithms scaled well and trends seemed to indicate they will continue to scale beyond the 16K processors tested.

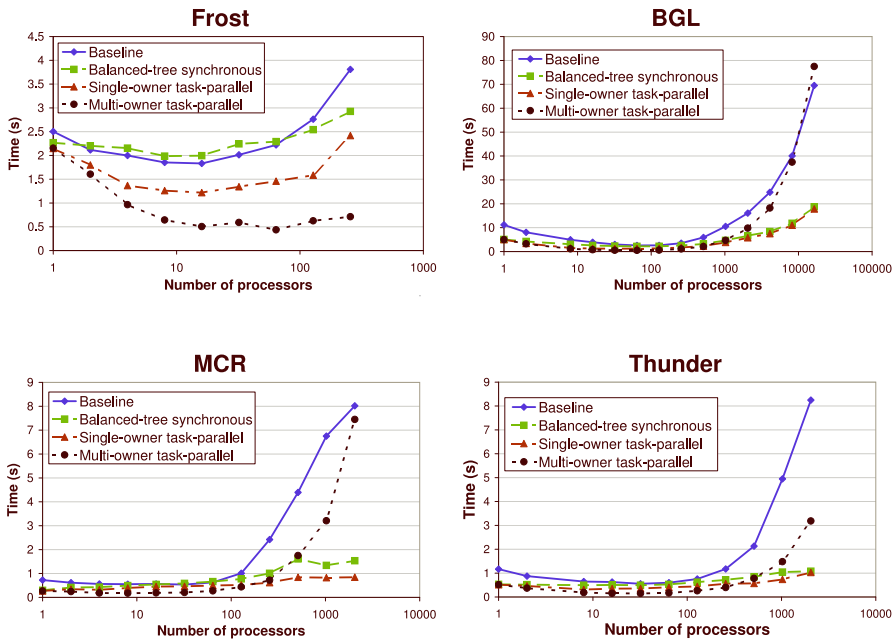


Figure 5: Clustering cost for advecting sinusoidal front problem. The line designated "baseline" is the implementation of reference [WHH03], while the others are three different modes of the new implementation.

Figure 6 breaks down the timings into the cost of clustering alone, and the cost of globalizing (accumulating and distributing) the results from all owners so all clustered boxes are known globally, by every processor. The "multi-owner task-parallel" mode is fastest for clustering alone on all four systems, but it required additional communication to gather the results among the multiple owners. The "single-owner task-parallel" mode maintained the result of each group on a single processor and avoided the more expensive all-gather communication. **Table 2 shows the increase in speed of the task-parallel modes relative to the baseline algorithm. If clustering and globalization are counted together, the single-owner mode is fastest and results in an increase in speed over the baseline algorithm by a factor of 3.4 to 9.5 at the highest number of processors on each platform. If we consider only the clustering step, without the globalizing step, the multi-owner mode is fastest and results in a 6.3- to 25-fold increase over the baseline.** Hence, the multi-owner mode is most efficient for clustering alone but the single owner case is fastest overall (figure 5), if the global set of boxes has to be distributed to every processor. The reason we make this distinction is that we are investigating approaches that would avoid having to make the global set of clustered boxes known globally and in this case the multi-owner approach may be more efficient.

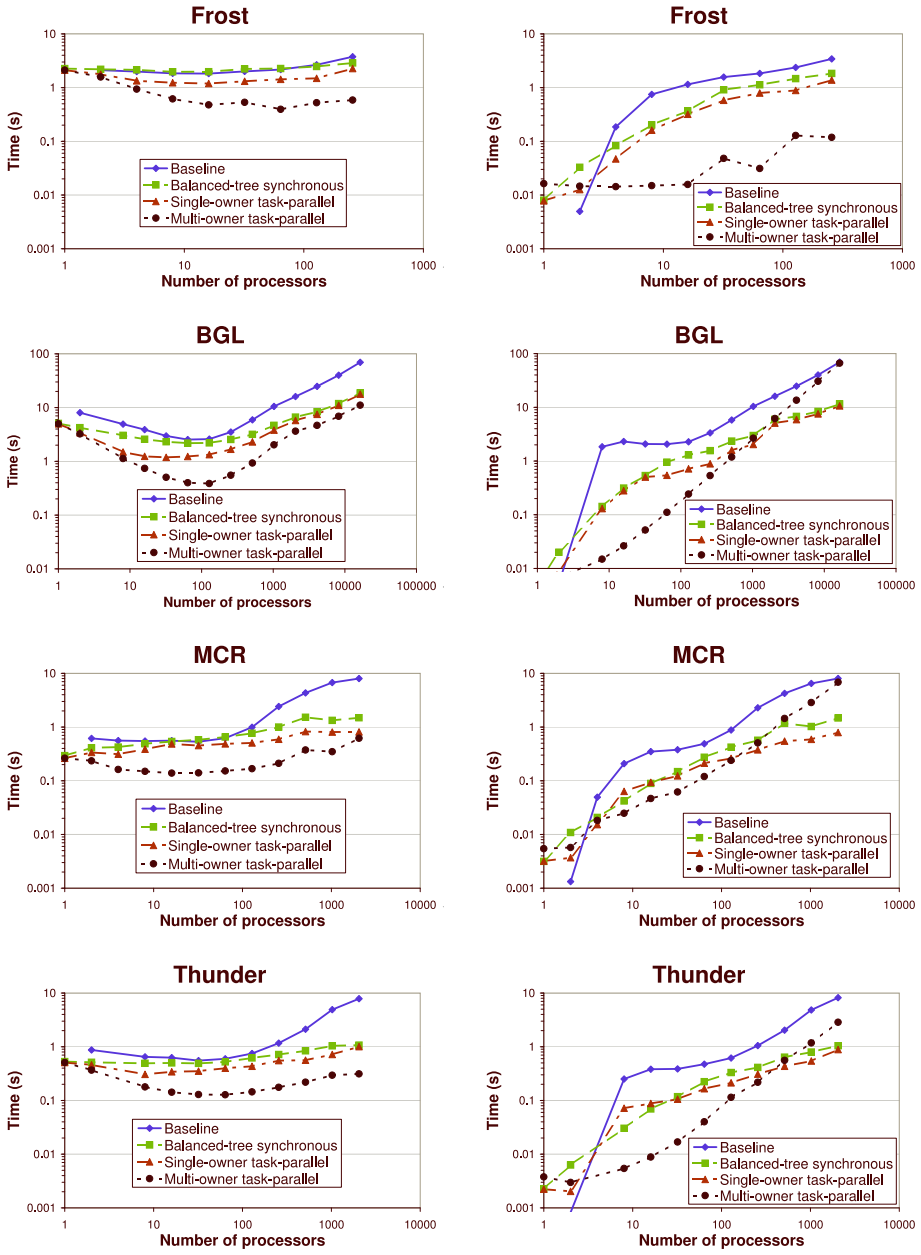


Figure 6: Isolated costs for the clustering (left column) and globally distributing the clustered results to all processors (right column). Globalizing times rises much faster than clustering time. The "multi-owner" task-parallel algorithm requires the least time for clustering on all four systems but requires additional costs for accumulating and distributing the results to all processors, because an all-gather is required among the multiple owners. The *total* times, clustering plus global accumulation and distribution, are shown in figure 5.

Algorithm steps	Task-parallel Mode	Machine and number of processors			
		Frost 256	BG/L 16K	MCR 2K	Thunder 2K
Cluster and globalize	Single-owner	1.6	3.4⋆	9.5⋆	8.0⋆
	Multi-owner	5.4⋆	0.9	1.1	2.6
Cluster only	Single-owner	1.7	3.9	9.8	7.8
	Multi-owner	14†	6.3†	13†	25†

Table 2: Speed-up factors for the task-parallel clustering algorithm on the largest number of processors of each platform, relative to the the speed of the baseline algorithm. The fastest cluster-and-globalize times are marked by a "⋆", and the fastest cluster-only times are marked by a "†". Note that the multi-owner mode was fastest on all systems when only clustering times were considered. The single-owner mode is fastest for most systems when both clustering and globalization are considered.

Note that the total times shown in figure 6 tend to be less than the sum of the two parts–clustering plus the globalization, shown in figure 5–because no explicit synchronization was performed between the two parts. Globalizing time may include time waiting for other processors to complete the clustering step. We are measuring maximum times (across all processors), and different quantities are at a maximum on different processors.

We next investigated the relative efficiency of the different approaches by breaking down, from the task manager's point of view, the time spent computing and the time spent waiting on messages. Computing generally consisted of computing the local signature, deciding whether and where to split a box, and determining groups and owners. Waiting from the task manager's point of view, meant waiting on communication, without a possibility of doing useful computation. (Wait time does not include waiting for children, as that time would include computing in the children tasks.) Figure 7 shows the computing and waiting times for the different modes. The multi-owner task-parallel mode was significantly faster than the other modes, both for computation and wait. This indicated the effectiveness of the work distribution approach used by the multi-owner mode. The single-owner modes spent equal time computing, as expected, because the most time spend computing is usually found on the single owner. However, the task-parallel "single-owner" mode spent about two-thirds less time waiting.

Overall, we found the the cost of clustering and the wait times for the new task-parallel algorithm, particularly the multi-owner case, scaled well. The imposed requirement that all processors know the resulting set of clustered boxes introduced a globalization step which is the main source of performance degradation for large numbers of processors. This requirement may be relaxed in the future as new approaches are developed that operate in a more localized manner.

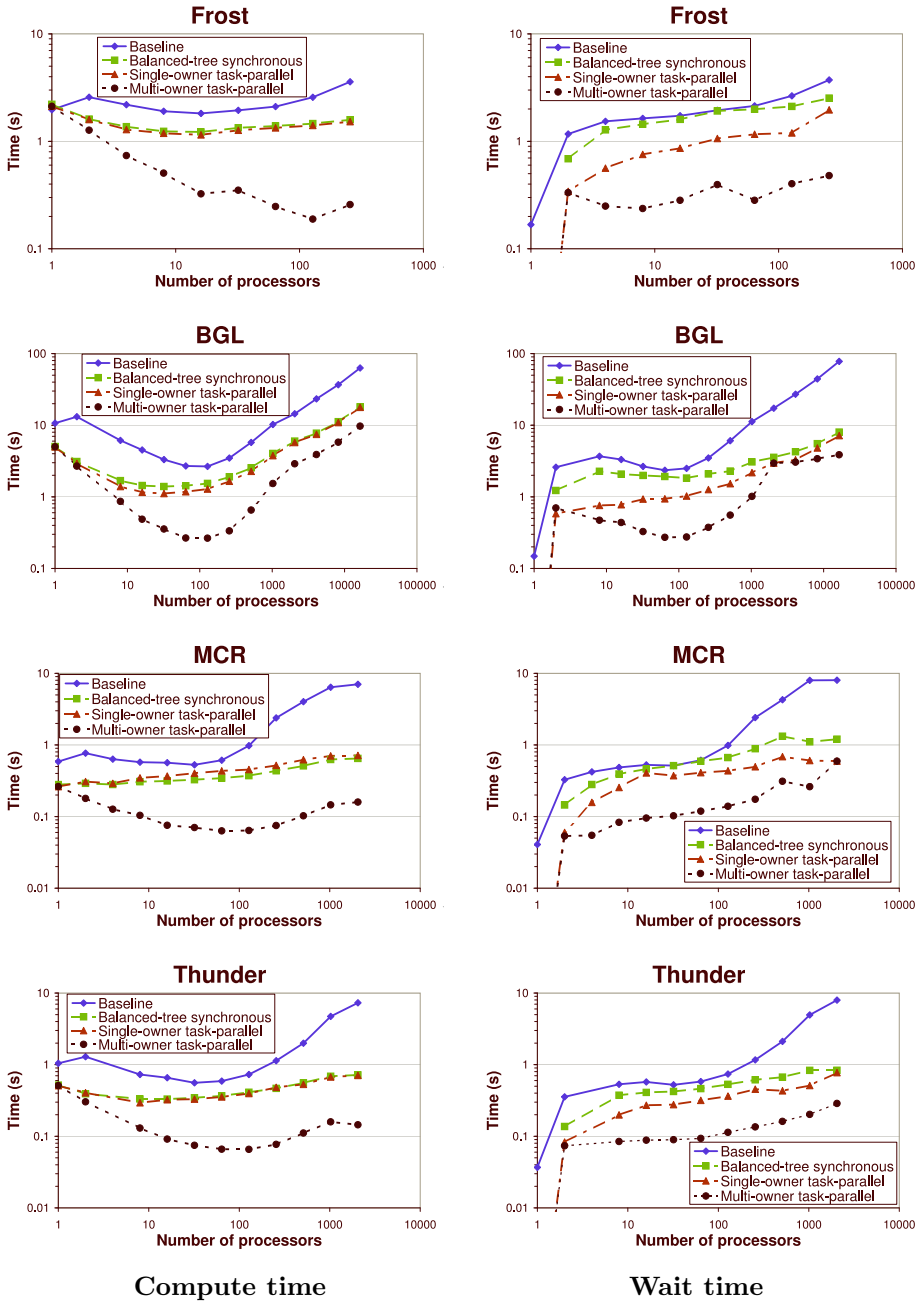


Figure 7: Time spent computing (left column) and waiting (right column) by the task manager during the clustering algorithm. The task manager time is the time to run the clustering algorithm; it does *not* include the globalization communication, which occurs *after* the clustering step. Although the "baseline" mode does not use a task manager, it has distinct computing and waiting phases that are equivalent to the task manager phases.

6 Summary

We presented a new task-parallel algorithm, based on the sequential top-down hierarchical clustering algorithm of Berger and Rigoutsos [BR91], which is commonly used to cluster cells in structured AMR calculations. The primary advantage of the new approach was that it generated many independent tasks that could be performed concurrently. Exploiting this on a parallel computer system resulted in better scalability of the algorithm on large-scale parallel platforms. We compared the performance of the new algorithm to a previous parallel clustering algorithm on several current parallel platforms, including up to 16K processors of the BlueGene/Light system. The new algorithm showed overall speed-up factors of 3.4 to 9.5 over the baseline algorithm.

The new algorithm achieved a high degree of parallelism by setting up independent thread-like tasks. Each task was itself an SPMD method, performing alternating computations and communications to generate a subset of the final output. Speed-up was attained by exploiting task-parallelism and overlapping the communications and computations of concurrent tasks. A task manager selected which task to work on, based on whether or not the task had all the information it needed to continue.

The algorithm was implemented within the much larger SAMRAI library. For the purposes of other grid generation steps, SAMRAI requires the boxes resulting from clustering to be known globally by every processor. Communicating the output boxes to all processors was a separate step that did not scale as well as the clustering step, especially when the clustering output was distributed over multiple processors.

The clustering operations in the new algorithm were most efficient when the workload was distributed over multiple processors, the so-called multi-owner mode. However, when run this way the result was distributed over the multiple processors requiring an all-gather communication to put the result on all processors. This all-gather step could be expensive. An alternative implementation created all the output on a single processor. In this case, the workload was not as well balanced, so the cost of the clustering operations increased, but the all-gather required in the multi-owner mode could be replaced by a simple broadcast so the globalization step is faster. Although the multiple-owner mode was the fastest clustering mode in our tests, when counting the globalizing step, the single-owner mode was usually faster. If other algorithms in SAMRAI could be reconfigured to avoid the globalization requirement, the multiple-owner implementation would be a faster alternative.

7 Acknowledgements

The authors gratefully acknowledge many useful discussions with Dr. David Hysom and the SAMRAI development team.

This work was performed under the auspices of the US Department of Energy by the University of California Lawrence Livermore National Laboratory under

contract No. W-7405-Eng-48. Report number UCRL-TR-207651.

References

[And73] Michael R. Anderberg. *Cluster Analysis for Applications*. Academic Press, New York, 1973.

[BB87] M. J. Berger and S. H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, 1987.

[BC89] M. J. Berger and P. Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics*, 82:65–84, 1989.

[BO84] Marsha J. Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computational Physics*, 53:484–512, 1984.

[BR91] M. Berger and I. Rigoutsos. An algorithm for point clustering and grid generation. *IEEE Transactions on Systems, Man and Cybernetics*, 21(5):1278–1286, September/October 1991.

[CGL+03] P. Colella, D. T. Graves, T. J. Ligocki, D. F. Martin, D. Modiano, D. B. Serafini, and B. Van Straalen. *Chombo Software Package for AMR Applications Design Document*. Applied Numerical Algorithms Group, NERSC Division, September 2003. http://seesar.lbl.gov/anag/chombo/ChomboDesign-1.4.pdf.

[DH73] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, New York, 1973.

[Har73] John A. Hartigan. *Clustering algorithms*. Wiley, New York, 1973.

[HK02] Richard D. Hornung and Scott R. Kohn. Managing application complexity in the SAMRAI object-oriented framework. *Concurrency and Computation: Practice and Experience*, 14:347–368, 2002.

[KB96] Scott R. Kohn and Scott B. Baden. Irregular coarse-grain data parallelism under LPARX. *Scientific Programming*, 5(3):185–201, 1996.

[MH80] D. Marr and E. Hildreth. Theory of edge detection. *Proceedings of the Royal Society of London*, 207, 1980.

[Ran03] Jarmo Rantakokko. An integrated decomposition and partitioning approach for irregular block-structured applications. In J. Rolim et. al., editor, *Parallel and Distributed Processing, 15 IPDPS 2000 Workshops, Cancun, Mexico, May 1-5, 2000, Proceedings*, pages 336–347, Berlin, June 2003. Springer-Verlag.

[RBL+00] Charles A. Rendleman, Vincent E. Beckner, Mike Lijewski, William Crutchfield, and John B. Bell. Parallelization of structured, hierarchical adaptive mesh refinement algorithms. *Computing and Visualization in Science*, 3(3):147–157, 2000.

[SAM04] SAMRAI web site, 2004. http://www.llnl.gov/CASC/SAMRAI/.

[WHH03] Andrew M. Wissink, David Hysom, and Richard D. Hornung. Enhancing scalability of parallel structured AMR calculations. In *Proceedings of the 17th ACM International Conference on Supercomputing (ICS03)*, pages 336–347, San Francisco, June 2003.